Stagefright: Scary Code in the Heart of Android

Researching Android Multimedia Framework Security

ZIMPERIUM MOBILE DEFENSE

Joshua "jduck" Drake
August 7th 2015
DEF CON 23

Agenda

- Introduction
- System Architecture
- Attack Surface
- Attack Vectors
- Vulnerability Discovery / Issues Found
- Exploitability / Mitigations
- Disclosure
- Conclusions

Introduction

About the presenter and this research

About Joshua J. Drake aka jduck

Focused on vulnerability research and exploit development for the past 16 years

Current Affiliations:

- Zimperium's VP of Platform Research and Exploitation
- Lead Author of Android Hacker's Handbook
- Founder of the #droidsec research group

Previous Affiliations:

- Accuvant Labs (now Optiv), Rapid7's Metasploit, VeriSign's iDefense Labs

Motivations

1. Improve the overall state of mobile security
    1. Discover and eliminate critical vulnerabilities
    2. Spur mobile software update improvements
2. Increase visibility of risky code in Android
3. Put the Droid Army to good use!

Sponsors

This work was sponsored by Accuvant Labs (now Optiv) with continuing support from Zimperium.

Special thanks go to Amir Etemadieh of Optiv / Exploiteers for his help with this research.

Additional thanks to Collin Mulliner and Mathew Solnik!

What is Stagefright?

- Android's Multimedia Framework library
  - written primarily in C++
- Handles all video and audio files
- Provides playback facilities - e.g. {Nu,Awesome}Player
- Extracts metadata for the Gallery, etc.

Brief History

- Android launched with an engine called OpenCORE
- Added to AOSP during Android Eclair (2.0) dev
- Optionally used in Android Froyo (2.2)
  - Both devices I have on 2.2 have it enabled
- Set as the default engine in Gingerbread (2.3) and later

- It's also used in Firefox, Firefox OS, etc.
  - first shipped in Firefox version 17
  - Used on Mac OS X, Windows, and Android
  - *NOT* used on Linux (uses gstreamer)

Why Stagefright?

1. Exposed via multiple attack vectors
    - **some of which require no user interaction**
2. Binary file format parsers are often vulnerable
    - Especially those written in native code
3. Various public mentions of instability (crashes)
    - /r/Android, AOSP bug tracker, googling for "mediaserver crash", etc.
4. Related publications about fuzzing the code

Related Work I

Fuzzing the Media Framework in Android (Slides)

by Alexandru Blanda and his team from Intel

- They released their tools! See: MFFA
- Interesting results!
  - tons of things reported
  - 7 accepted as security issues
  - 3 fixed in AOSP
- CVE-2014-7915, CVE-2014-7916, CVE-2014-7917

MORE ON THESE LATER ;-)

Related Work II

On Designing an Efficient Distributed Black-Box Fuzzing System for Mobile Devices

by Wang Hao Lee, Murali Srirangam Ramanujam, and S.P.T. Krishnan of Singapore's Institute for Infocomm Research

- Focused on tooling more than bugs
- Not focused on Android only
- Found several bugs, but analysis seems lacking/incorrect
- Unclear if any issues were fixed as a result

Related work

Pulling a John Connor: Defeating Android

by Charlie Miller at Shmoocon 2009

- Discusses fuzzing a media player
  - got crashes in mediaserver
- Focused on opencore, not Stagefright
- Focused on pre-release G1
- Really old, research done in 2008

However, due to apparent lack of proactive Android security research it seems relevant still.

About this research

Stagefright is big and supports a wide variety of multimedia file formats.

Rather than dividing my focus among multiple formats, I focused on **MPEG4**.

This allowed me to be more thorough in eliminating issues.

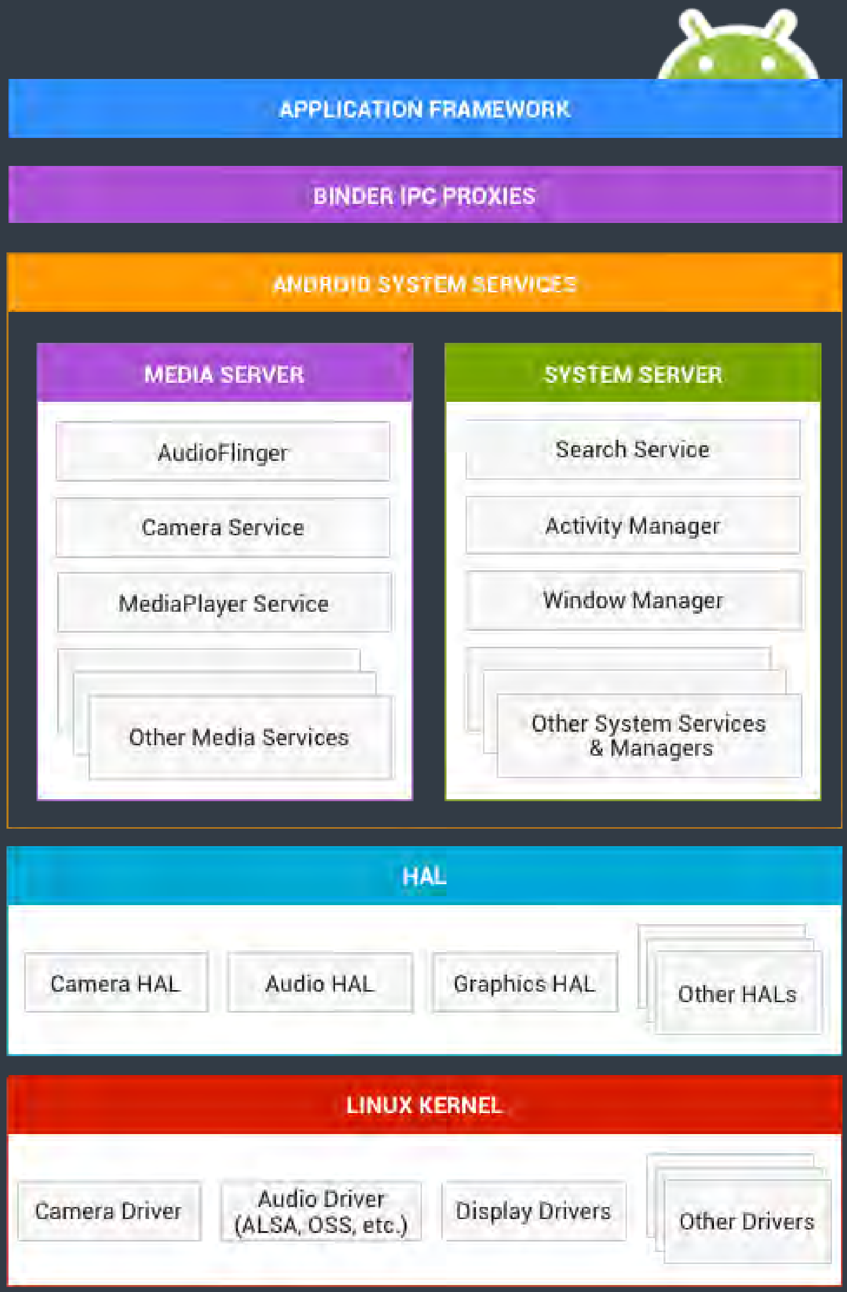
As such, the rest of this presentation will be somewhat specific to Stagefright's MPEG4 processing.

System Architecture

Processes, privileges, etc.

Android Architecture

- Android is very modular
  - Things run in separate processes
  - Lots of inter-process communications
- "Sandbox" relies on modified scheme based on Linux users and groups
- libstagefright executes inside "MEDIA SERVER"



Picture from Android Interfaces in the Android Developer documentation

Process Architecture

The *mediaserver* process runs in the background:

```
media     181   1     120180 10676 [...] S /system/bin/mediaserver
```

It's a native service that's started at boot from */init.rc*:

```
service media /system/bin/mediaserver
    class main
    [...]
```

As such, the process automatically restarts when it crashes.

Process Privileges (Nexus 5)

The last part of the service definition in */init.rc* shows the privileges that the service runs with:

```
service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc mediadrm
```

WHOA! This service is very **PRIVILEGED**!

Android apps *CANNOT* request/receive permissions like *audio*, *camera*, *drmrpc*, and *mediadrm*

But there's more...

mediaserver Privilege Survey

A Droid Army provides quick and valuable survey results!!

I surveyed 51 devices. The breakdown by OEM was:

```
Count  $(BRAND)
====== ====================
17     Nexus/Google
13     Motorola
9      Samsung
6      HTC
3      LG
1      Sony
1      Amazon
1      ASUS
1      Facebook
1      OnePlus/Cyanogen
1      SilentCircle/SGP
```

Let's look at accessible groups, sorted by # of devices...

Privilege Survey Results I

```
CNT GROUP                  PURPOSE
51  3003(inet)             /* can create AF_INET and AF_INET6 sockets */
51  3002(net_bt)           /* bluetooth: create sco, rfcomm or l2cap sockets */
51  3001(net_bt_admin) /* bluetooth: create any socket */
51  1006(camera)           /* camera devices */
51  1005(audio)            /* audio devices */
[...]
```

All devices had this level of access, with which you can:

- Monitor, record, and playback audio
- Access camera devices
- Connect to hosts on the Internet
- Access and configure bluetooth

Ouch! This allows an attacker to spy on you already.

Privilege Survey Results II

Continuing down the line, things get interesting...

```
CNT GROUP               PURPOSE
33  3007(net_bw_acct)   /* change bandwidth statistics accounting */
33  1026(drmrpc)        /* group for drm rpc */
27  1000(system)        /* system server */
20  1003(graphics)      /* graphics devices */
19  1031(mediadrm)      /* MediaDrm plugins */
18  3004(net_raw)       /* can create raw INET sockets */
11  3009(qcom_diag)     /* <jduck> baseband debugging? */
9   1028(sdcard_r)      /* external storage read access */
8   1023(media_rw)      /* internal media storage write access */
8   1004(input)         /* input devices */
7   1015(sdcard_rw)     /* external storage write access */
4   2000(shell)         /* adb and debug shell user */
4   1001(radio)         /* telephony subsystem, RIL */
```

and more!

Architecture Recap

To recap the important bits...

1. libstagefright processes media inside *mediaserver*
2. The service runs privileged, potentially even as "system"
3. *mediaserver* automatically restarts

The additional attack surface exposed to a compromised *mediaserver* is large — even compared to ADB. Beware.

Attack Surface

Where is the code under attack?

Locating the Attack Surface

Once you have your environment set up, finding the MPEG4 attack surface is relative straight-forward.

1. Attach debugger to mediaserver process
2. Place breakpoint on *open*
3. Open an MPEG4 video file
4. Sift through breakpoint hits until *r0* points at your file
5. Look at the backtrace
6. Dig in and read the surrounding code

NOTE: Released tools include some helper scripts.

What do you find?

```
[*] open("/sdcard/Music/playing.mp4",...) called from:
#0  open (pathname=<value optimized out>, flags=0) at bionic/libc/unistd/open
#1  0x40b345e8 in FileSource (this=0x479038, filename=0x478d08 "/sdcard/Music
#2  0x40b332fe in android::DataSource::CreateFromURI (uri=0x478d08 "/sdcard/M
#3  0x40b2ef50 in android::AwesomePlayer::finishSetDataSource_l (this=0x47805
 at frameworks/base/media/libstagefright/AwesomePlayer.cpp:2085
#4  0x40b2efb2 in android::AwesomePlayer::onPrepareAsyncEvent (this=<value op
#5  0x40b2c990 in android::AwesomeEvent::fire (this=<value optimized out>, qu
#6  0x40b50c28 in android::TimedEventQueue::threadEntry (this=0x47806c) at fr
#7  0x40b50c6c in android::TimedEventQueue::ThreadWrapper (me=0x47806c) at fr
#8  0x400e8c50 in __thread_entry (func=0x40b50c59 <android::TimedEventQueue::
#9  0x400e87a4 in pthread_create (thread_out=<value optimized out>, attr=0xbe
```

frame #3 - **frameworks/base** / **media/libstagefright/AwesomePlayer.cpp:2085**

(note: moved to frameworks/av in Android >= 4.1)

AwesomePlayer.cpp:2085

```
2085        dataSource = DataSource::CreateFromURI(mUri.string(), &mUriHeade
2086    }
....
2092    sp<MediaExtractor> extractor;
2093
2094    if (isWidevineStreaming) {
....
2109    } else {
2110        extractor = MediaExtractor::Create(
2111                dataSource, sniffedMIME.empty() ? NULL : sniffedMIME.c_s
....
2127    status_t err = setDataSource_l(extractor);
```

Okay, so it calls *setDataSource_l(sp<MediaExtractor>)*...

Let's look at that.

AwesomePlayer::setDataSource_l

```
349 status_t AwesomePlayer::setDataSource_l(const sp &extractor) {
...
356     for (size_t i = 0; i < extractor->countTracks(); ++i) {
```

... calls *MPEG4Extractor::countTracks*:

```
305 size_t MPEG4Extractor::countTracks() {
...
307     if ((err = readMetaData()) != OK) {
```

In turn, that calls *readMetaData*. Let's check that out...

```
365 status_t MPEG4Extractor::readMetaData() {
...
372     while ((err = parseChunk(&offset, 0)) == OK) {
373     }
```

readMetaData calls *parseChunk*. Let's look at that!

MPEG4Extractor::parseChunk

This function is the primary attack surface for MPEG4 parsing!

- primary dispatch for handling MP4 atoms / FourCC values
  - between 80 and 140 depend on Android version
- it's implemented using recurison

```
    switch(chunk_type) {
        case FOURCC('m', 'o', 'o', 'v'):
        case FOURCC('t', 'r', 'a', 'k'):
...
            while (*offset < stop_offset) {
                status_t err = parseChunk(offset, depth + 1);
```

More specific examples will follow in later sections.

Attack Vectors

What would an attack look like?

Vector Enumeration Methodology

Ultimate goal: Find out how to get attacker controlled media files processed by this code.

- Try all possible ways to send yourself a media file!
- Depends on knowledge of "all possible ways"

A Thorough Methodology:

1. Find all calls into this function.
2. Ask yourself "Can an attacker's data reach here?"
3. Repeat until all vectors are identified.

Modularity Complicates Matters

Executing the thorough methodology is challenging due to:

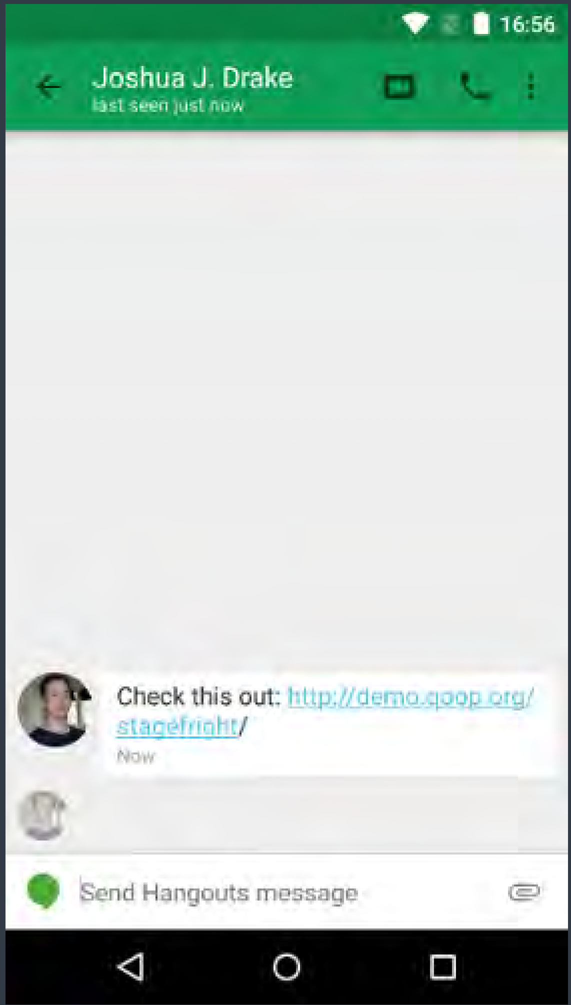
- A mix of Java and native code
- Object-oriented (OO) code
- Must be mindful of member objects & instantiation
- Code paths traverse a variety of *Service* and *BroadcastReceiver* end points
- Some vectors might be closed source (e.g. Google apps)

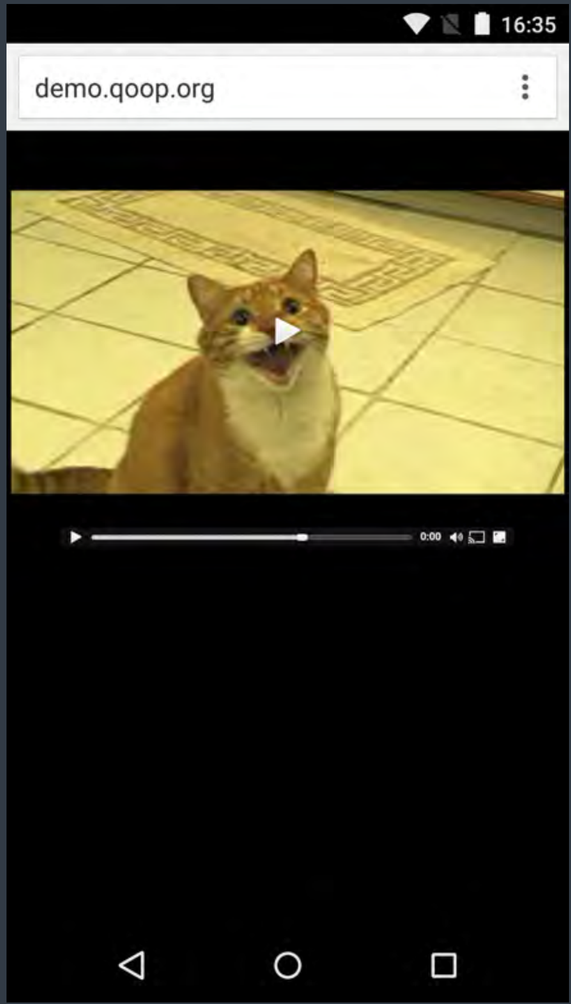
IMHO this is still the best way to learn "all possible ways".

Vector I: Media in the Browser

The <video> tag is new in HTML5! Let's try it...

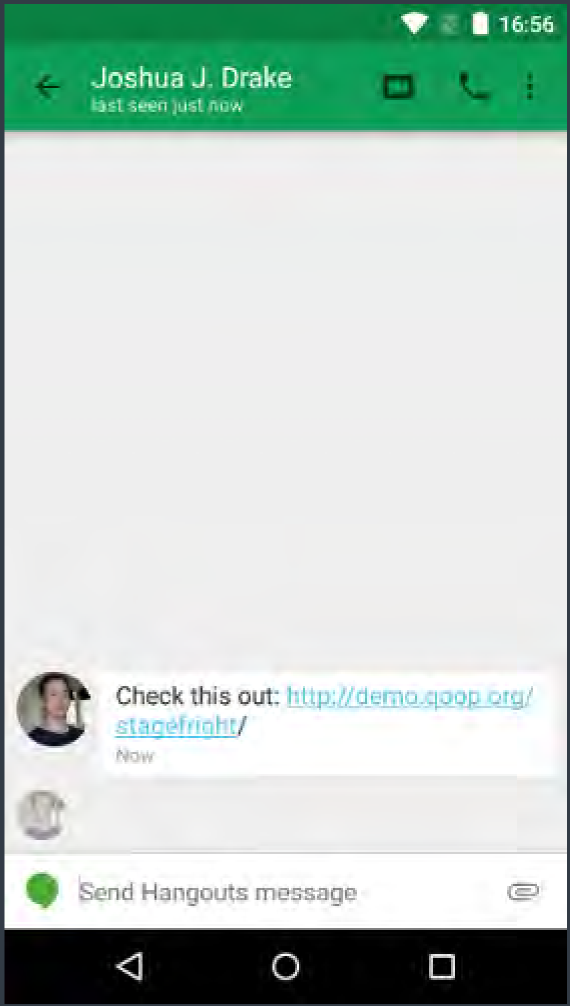
...Yep, it works!

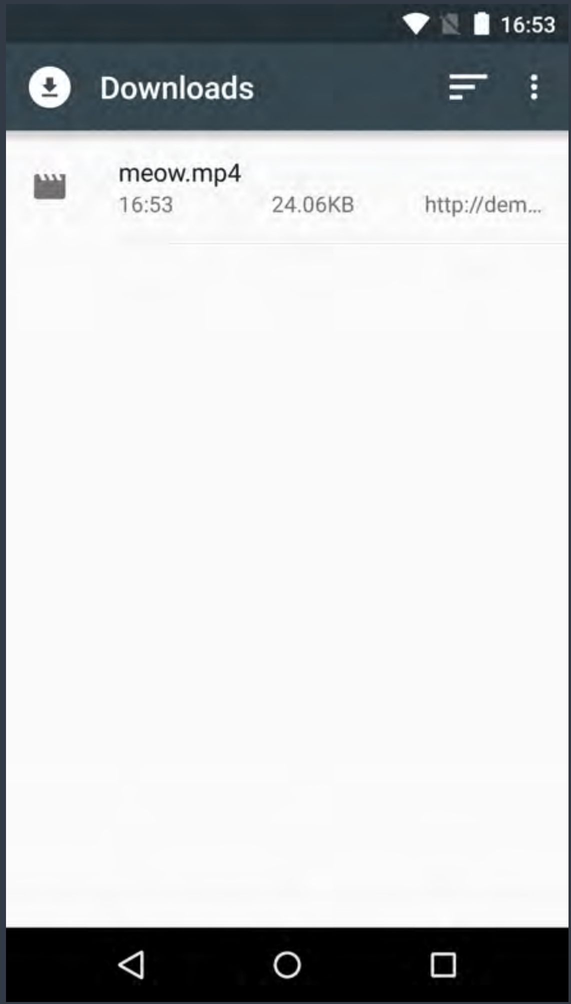




Vector II: Browser Auto-download

Also, servers can force you to download instead!





Vector II: Browser Auto-download II

- Downloads happen in the background.
- No prompting to the user.
  - No option to prompt either :-/
    - FEATURE REQUEST!
  - This behavior has been abused in the past...
    - Thomas Cannon's "Data Stealing"
    - Not just on Android! "Carpet Bombing" attack

Testing shows it processes media when it's finished downloading!

See also: http://developer.android.com/reference/android/app/DownloadManager.html

Enter the Media Scanner

After a long journey looking into browser download processing, I discovered the *MediaScanner*, which:

- Extracts metadata for the Gallery and so on.
- Is invoked in various ways, including:
  - Directly, via *MediaScannerConnection*
  - MEDIA_{MOUNTED,SCANNER_SCAN_FILE} Intents
  - Classes implementing *MediaScannerConnectionClient*

With our new understanding, we continue our methodology and track backwards to untrusted data sources. We find...

NOTE: For more details, see the bonus slides or whitepaper (in progress).

Tons of Attack Vectors!

We find a multitude of attack vectors that use Stagefright!

In summary, any way your device touches media:

- Mobile Network - Mms
- Client Side - Browser, Downloads, Email
- Physically Adjacent - Nfc, Bluetooth, VCards
- Physical - SD Cards, USB OTG Drives, USB MTP/PTP
- Misc - Gallery

Total attack vectors: **11+**

Do you use any of these to talk to untrusted people?

The Scariest Part - MMS

- Media is **AUTOMATICALLY** processed **ON MMS RECEIPT**.
- **BEFORE** creating a notification!
  - Actually, while creating the notification

Exploiting a vulnerability in Stagefright via MMS could allow **SILENT, REMOTE, PRIVILEGED** code execution.

- The attacker's payload simply needs to prevent the notification.

Who has your phone number?

Where does this work?

- Works in latest version of Hangouts
  - The default MMS application in 5.0+
  - Google removed com.android.mms

- Works in latest version of Messenger
  - Popular alternative to Hangouts
  - Now com.google.android.apps.messaging

- **TURN AUTO-RETRIEVE OFF!**
  - Not a silver bullet, 10+ vectors left...

Doesn't seem to work in com.android.mms (AOSP:packages/apps/Mms)

Triggers Virally

The vulnerable code is invoked many times in Android.

- Basically any time a thumbnail is rendered or metadata is needed (e.g. dimensions)
  - Rotating the screen
  - Starting the Messaging app (conversation list)
  - Viewing the Gallery
  - Sharing malicious media
  - and so on...

Any Other Vectors?

There could be additional vectors! Consider:

- Downstream (OEM/Carrier) modifications
- Third-party apps

Untested ideas:

- Instant messaging?
- Social networks?
- QR Codes?

Please reach out if you have ideas or discover additional vectors!

Vulnerability Discovery

Are there security bugs in Stagefright?

Discovery Methodology

This is the basic methodology I used for the first pass:

1. Write fuzzer (basic dumb fuzzer in this case)
2. Run the fuzzer
3. While fuzzer runs, read code
4. When fuzzer finds crashes, read surrounding code
5. Repeat until brain melted

First Round Specifics

Again, the decision was to focus on MP4 video.

- Seemed complicated enough...
- Had the most lines of code
- Same code handles other formats (3GP, M4A)

Corpus

- What code your inputs exercise matters
- Didn't even bother with building an optimized set
- Started with the smallest file possible
- @Zenofex created *meow.3gp* - 25KB

First Round Results

The fuzzer ran on live Android devices for ~1 week.

- Results: ~6200 crashes
- Most crashes not interesting
- Post-analysis results: ~20 unique bugs
  - None of these were very interesting

However, code review during analysis was fruitful!

Found ~5 memory corruptions nearby during code review
These became CVE-2015-1538 and CVE-2015-1539

Enter American Fuzzy Lop

AFL is a coverage-guided fuzzer that gravitates towards new code paths.

- Useful for generating a corpus
- Able to find buggy code paths quickly

Second round methodology:

- Develop harness to test Stagefright
- Run AFL on beefy hardware
- Periodically triage, analyze, and restart the fuzzer
- Catalog and fix bugs as they are discovered

See http://lcamtuf.coredump.cx/afl/

Second Round Results

I ran the second round of testing for about 3 weeks.

- Used both default and dictionary based modes
- Tried with and without ASAN
- ~3200 tests per second
- Total CPU hours was over 6 months

Five more critical issues discovered!

Plenty more less-severe crashing bugs too..

The code fuzzed clean at the end.

Bug Summary

CVE-2015-1538 #1 -- MP4 'stsc' Integer Overflow
CVE-2015-1538 #2 -- MP4 'ctts' Integer Overflow
CVE-2015-1538 #3 -- MP4 'stts' Integer Overflow
CVE-2015-1538 #4 -- MP4 'stss' Integer Overflow
CVE-2015-1539 ------ MP4 'esds' Integer Underflow
CVE-2015-3824 ------ MP4 'tx3g' Integer Overflow
CVE-2015-3826 ------ MP4 3GPP Buffer Overread
CVE-2015-3827 ------ MP4 'covr' Integer Underflow
CVE-2015-3828 ------ MP4 3GPP Integer Underflow
CVE-2015-3829 ------ MP4 'covr' Integer Overflow
..and a whole slew of stability fixes

Details for a FAIL

Due to time constraints, let's look at a few interrelated issues found in round 1.

Fixes pushed to AOSP in Lollipop release:

```
Date:   Mon Jul 28 09:54:57 2014 -0700

    SampleTable: check integer overflow during table alloc

    Bug: 15328708
    Bug: 15342615
    Bug: 15342751
```

Full vulnerability analysis details will be published in the whitepaper (in progress)

Three Related Issues

All three are very similar, so let's look at just one:

```
@@ -330,6 +330,10 @@ status_t SampleTable::setTimeToSampleParams(
     }

     mTimeToSampleCount = U32_AT(&header[4]);
+    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
+    if (allocSize > SIZE_MAX) {
+        return ERROR_OUT_OF_RANGE;
+    }
     mTimeToSample = new uint32_t[mTimeToSampleCount * 2];

     size_t size = sizeof(uint32_t) * mTimeToSampleCount * 2;
```

Okay. So if the 64-bit result is bigger than 2^32, we return *ERROR_OUT_OF_RANGE*.

Right?

Embarrassing, but Educational

SORT OF. So what REALLY happens?

> *In C, on the other hand, if you multiply two 32-bit integers, you get a 32-bit integer again, losing the upper 32 bits of the result. ... which is a typical mistake of inexperienced C hackers.*

All three multiplicands are *uint32_t*
No integer promotion so upper 32-bits are lost
No integer overflow is ever detected.
The original vulnerability remained. OOPS.

"Catching Integer Overflows in C", Felix von Leitner, http://www.fefe.de/intof.html

Can these issues be exploited?

```
mSampleToChunkEntries =
    new SampleToChunkEntry[mNumSampleToChunkOffsets];
```

```
mSampleToChunkEntries =
    malloc( mNumSampleToChunkOffets * sizeof(SampleToChunkEntry) )
```

DEMO!

What about getting these issues fixed?

Wait, what are you trying to say?

Thanks for your time!

Any questions?

Prefer to ask offline? Contact me:

Joshua J. Drake
jdrake@zimperium.com
jduck @ Twitter/IRC
www.droidsec.org

BONUS SLIDES!!!

Be sure to thank me for the extra content =)

Discovering the Media Scanner

Discovering the Media Scanner

Looking at the Browser's *DownloadHandler* is the beginning of a journey down the rabbit hole.

```
/*
 * Handle download requests
 */
public class DownloadHandler {
...
    /*package */ static void onDownloadStartNoStream(Activity activity,
...
        final DownloadManager.Request request;
        try {
            request = new DownloadManager.Request(uri);
...
        // let this downloaded file be scanned by MediaScanner -
        // so that it can show up in Gallery app, for example.
        request.allowScanningByMediaScanner();
```

DownloadManager.Request.allowScanningByMediaScanner

Media Scanner II

But how does that work?!

To see, we consult DownloadManager.java in frameworks/base/core/java/android/app:

```
        public void allowScanningByMediaScanner() {
            mScannable = true;
        }
```

shrug

Let's try again with *mScannable*...

Media Scanner III

And looking into *mScannable*, we find:

```
    public static class Request {
...
        private boolean mScannable = false; // THANKFULLY
...
         * @return ContentValues to be passed to DownloadProvider.insert
         */
        ContentValues toContentValues(String packageName) {
...
            // is the file supposed to be media-scannable?
            values.put(Downloads.Impl.COLUMN_MEDIA_SCANNED, (mScannable)
                    SCANNABLE_VALUE_NO);
```

Alright, so now we are going off to *DownloadProvider*...

Having fun yet?

MediaScanner IV

DownloadProvider is a *Service* that processes a queue of files to download. The most relevant part of the code follows:

```
public class DownloadService extends Service {
...
    private DownloadScanner mScanner;
...
    /**
     * Update {@link #mDownloads} to match {@link DownloadProvider} state
     * Depending on current download state it may enqueue {@link Download
     * instances, request {@link DownloadScanner} scans, update user-visi
...
    private boolean updateLocked() {
...
            // Kick off download task if ready
            final boolean activeDownload = info.startDownloadIfReady(mExecuto

            // Kick off media scan if completed
            final boolean activeScan = info.startScanIfReady(mScanner);
```

MediaScanner V

Looking closer at *DownloadScanner*, we see:

```
public class DownloadScanner implements MediaScannerConnectionClient {
...
        public void exec(MediaScannerConnection conn) {
            conn.scanFile(path, mimeType);
        }
```

This sends us down another rabbit hole, to see the internals of the MediaScanner implementation. More details on that will be in the whitepaper.

Suffice to say that it eventually leads to *MPEG4Extractor::parseChunk*.

MediaScanner VI

Stepping back, we see that another API that leads to scanning too...

DownloadManager.addCompletedDownload (since API 12)

```
frameworks/base/core/java/android/app/DownloadManager.java:1199: \
     return addCompletedDownload(title, description, isMediaScannerScannable,
          mimeType, path,
frameworks/base/core/java/android/app/DownloadManager.java-1200- \
          length, showNotification, false);
```

Let's look into calls to this API and see if they do or don't scan things.

MediaScanner VII

These don't scan media:

```
packages/apps/Browser/src/com/android/browser/Controller.java:2118: \
              manager.addCompletedDownload(target.getName(),
packages/apps/Browser/src/com/android/browser/Controller.java-2119- \
                     mActivity.getTitle().toString(), false,

packages/apps/Email/emailcommon/src/com/android/emailcommon/utility/Attachmen
             long id = dm.addCompletedDownload(attachment.mFileName, attachme
packages/apps/Email/emailcommon/src/com/android/emailcommon/utility/Attachmen
                     false /* do not use media scanner */,

packages/providers/DownloadProvider/src/com/android/providers/downloads/Downl
104     @Override
105     public String createDocument(String docId, String mimeType, String di
...
         return Long.toString(mDm.addCompletedDownload(
packages/providers/DownloadProvider/src/com/android/providers/downloads/Downl
                 file.getName(), file.getName(), false, mimeType, file.
```

MediaScanner IV

These DO use the media scanner:

```
packages/apps/UnifiedEmail/src/com/android/mail/providers/EmlAttachmentProvid
                    mDownloadManager.addCompletedDownload(attachment.getName(),
packages/apps/UnifiedEmail/src/com/android/mail/providers/EmlAttachmentProvid
                            description, true, attachment.getContentType(),
```

After reading some documentation and searching around for more details about the MediaScanner, we see that it can also be triggered via several *Intent*s.

- android.intent.action.MEDIA_MOUNTED
- android.intent.action.MEDIA_SCANNER_SCAN_FILE

Vectors into the Media Scanner

Vectors into the Media Scanner I

Users of *MediaScannerConnection* include:

- The Android Compatability Test Suite (CTS)
- The *ExternalStorage* sample in *ApiDemos*
- The Roboelectric test suite
- The *CameraBrowser*'s *ObjectViewer*
- *CarouselViewUtilities* (??)
- *BluetoothOppService*
- *VCardService*
- Email app *AttachmentUtilities*
- The Gallery (of course) *IngestService*
- ....and....

Vectors into the Media Scanner II

Users of *MediaScannerConnection* also include:

- Nfc app *HandoverTransfer*
- *CalendarProvider*'s *CalendarDebugActivity*
- *DownloadProvider*'s *DownloadScanner*
    - used by the Browser, via *DownloadManager*
- *MediaProvider*
    - Implements Intents for scanning
- *TestingCamera* from the PDK

It's important to note that some vectors don't process untrusted data. (i.e. the Camera and test suites)

Vectors into the Media Scanner III

Locations that invoke via the MEDIA_MOUNTED Intent include:

- The *MediaScannerActivity* sample
- *MountService* (via *vold*)
- Music app *TestSongs*

This includes when SD cards are inserted as well as when dealing with MTP connections.

Vectors into the Media Scanner IV

Locations that invoke via the MEDIA_SCANNER_SCAN_FILE Intent include:

- Taking pictures from within the Browser (*SelectFileDialog* or *UploadHandler*)
- The *screenrecord* command
- "photobasics"
- Mms app *ComposeMessageActivity*
  - Ringtones and Media via *copyPart*
- SoundRecorder app *SoundRecorder*
- UnifiedEmail app *EmlAttachmentPRovider*
- VideoEditor app *ApiService*

Vectors into the Media Scanner V

Classes that implement the *MediaScannerConnectionClient* interface include:

- The Android CTS
- *CameraBrowser.ObjectViewer*
- Bluetooth app *BluetoothOppService*
- Contacts app *VCardService*
- Gallary2 app
- *DownloadProvider.DownloadScanner*
- *MediaProvider*

h0dg3 p0dg3

Caveats to Attacking via MTP/PTP

- MTP/PTP requires a USB connection
- It's enabled by default on Nexus devices since 4.0
  - Can be disabled (mine is)
    - Can't disable it on some devices (i.e. SGS5) :-(
- Requires an unlocked while USB is plugged in!
  - Doesn't apply to "None" or "Swipe" screen locking

Sending MMS w/o Carriers I

- need to broadcast WAP_PUSH_RECEIVED
  - can't do it via "am broadcast"
    - it doesn't support *byte[]* Intent extras
  - inject a re-broadcast receiver (MmsProxy) into com.android.phone with adbi/ddi
- MMSC connections forced over mobile network
  - netd adds a route temporarily
  - created a patch to netd to avoid that

Sending MMS w/o Carriers II

Modify APN settings

- remove "mmsc" from existing APN
- create new APN with:
  - LAN server for MMSC
  - "mmsc" in APN type
- host your own MMSC

Now you're ready to test!